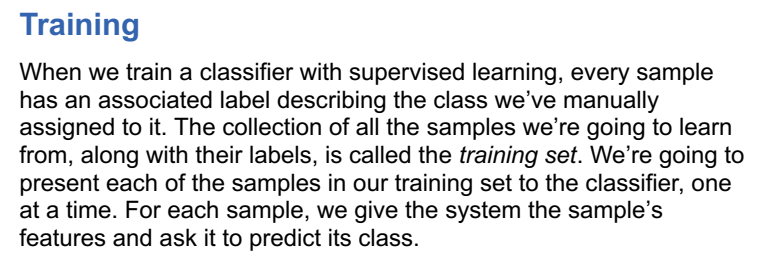
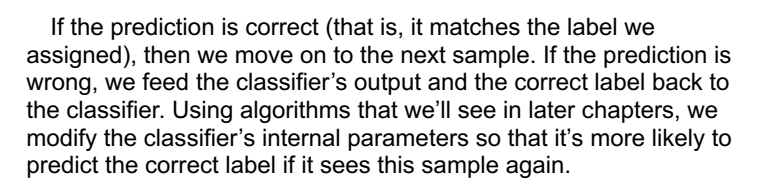
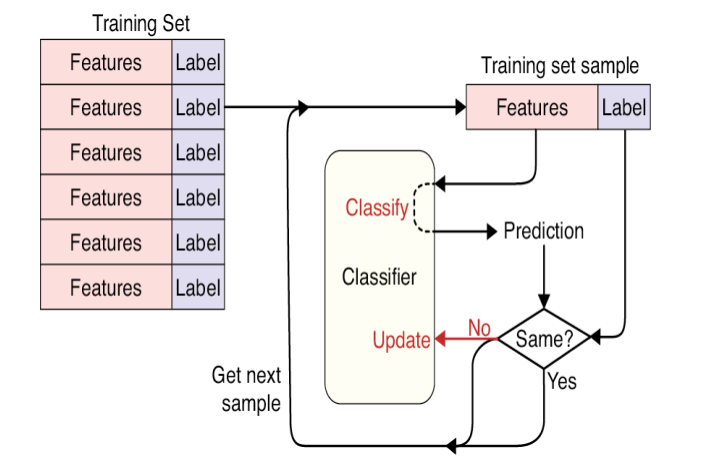
**Training**, the process of taking a system that’s been  
initialized with default or random values and gradually improving it so that it’s tuned to the data we want to  
understand.



****

****

****

As our training process runs through this loop, one sample at a  
time, the classifier’s internal variables are nudged toward values that  
do an increasingly good job of predicting labels.

Each time we run through the entire training set, we say that we’ve trained for one ***epoch*.**

We usually run the system through many epochs so the system sees every sample many times.

Typically, we keep training as long as the system is still learning and improving its performance on the training data, but we might stop if we run out of time, or if we run into problems.

The **training process** works as follows

First, the weights of the network are initialized to small random values. Then the network performs a series of training steps.

At each training step, one batch of images is passed through the network and the errors are back propagated to update the weights.

The batch size determines how many images are in each training step batch.

The larger the batch size, the more stable the gradient calculation, but the slower each training step.

It would be far too time-consuming and computationally intensive to use the entire dataset to calculate the gradient at each training step, so generally a batch size between 32 and 256 is used.

It is also now recommended practice to increase the batch size as training  
progresses.  
This continues until all observations in the dataset have been seen once. This completes the first epoch.

The data is then passed through the network again in batches as  
part of the second epoch.

This process repeats until the specified number of epochs have elapsed.

**Training Deep Neural Nets**

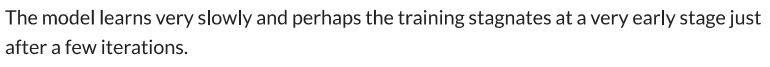
You may need to train a much deeper DNN, perhaps with (say) 10 layers, each containing hundreds of neurons, connected by hundreds of thousands of connections.   
1. First, you would be faced with the tricky *vanishing gradients* problem (or the related *exploding gradients* problem) that affects deep neural networks and makes lower layers very hard to train.  
2. Second, with such a large network, training would be extremely slow.  
3. Third, a model with millions of parameters would severely risk overfitting the training set.



Back propagation algorithm works by going from the output layer to the  
input layer, propagating the error gradient on the way.

Once the algorithm has computed the gradient of the cost function with regards to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent step.

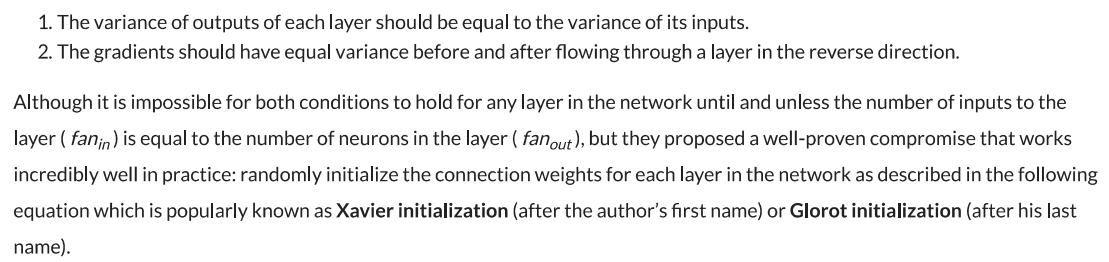


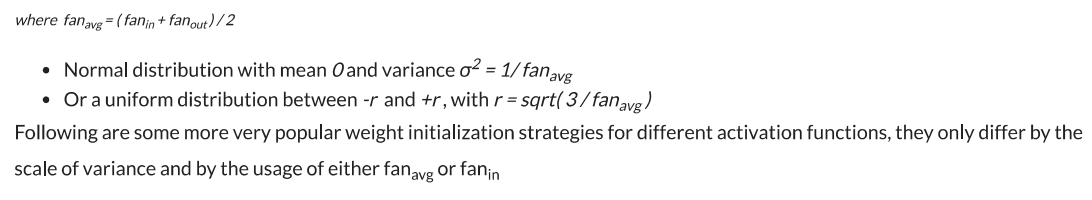




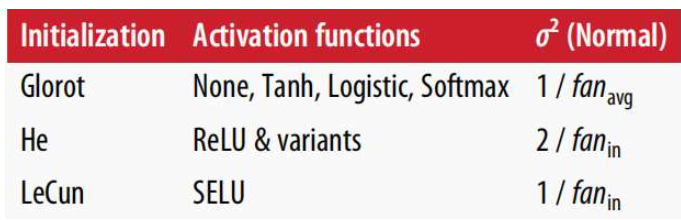




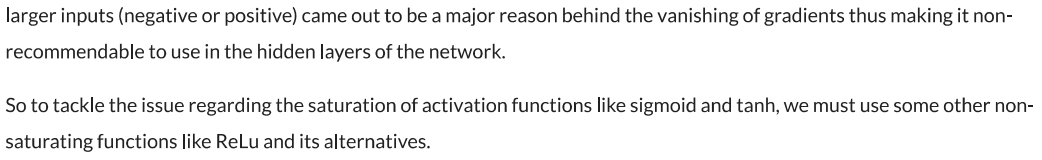


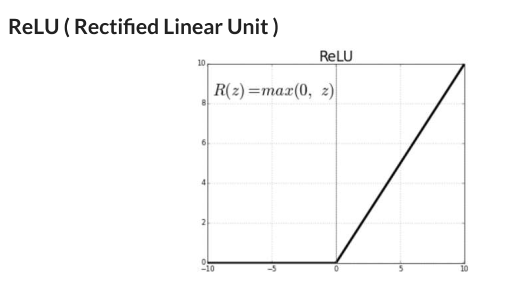


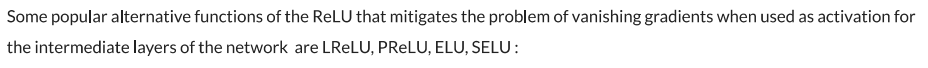


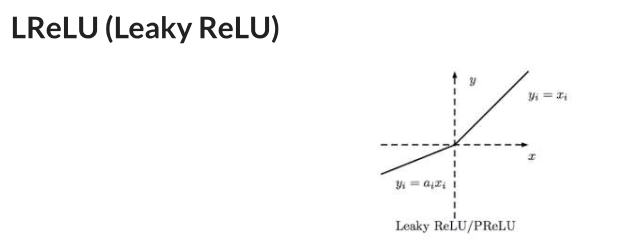


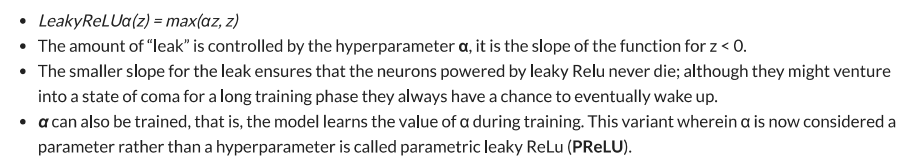


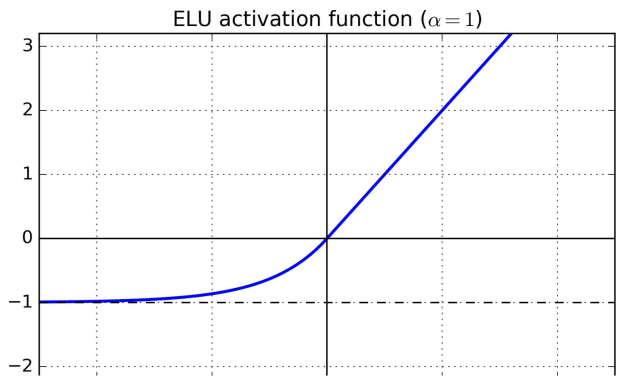


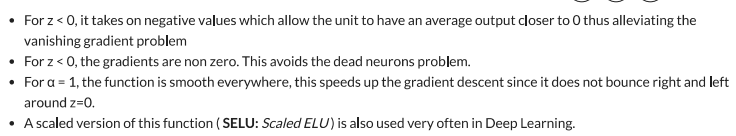


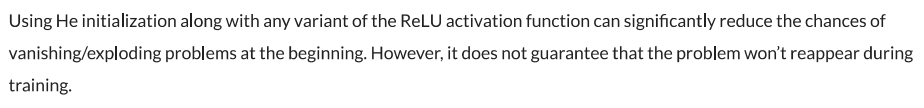


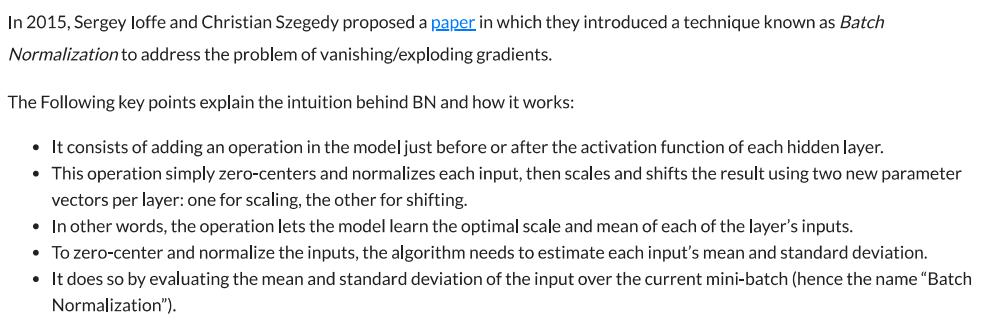












**Batch Normalization**One common problem when training a deep neural network is ensuring that the weights of the network remain within a reasonable range of values—if they start to become too large, this is a sign that your network is suffering from what is known as the exploding gradient problem.

As errors are propagated backward through the network, the calculation of the gradient in the earlier layers can sometimes grow exponentially large, causing wild fluctuations in the weight values.

If your loss function starts to return NaN, chances are that your weights have grown large enough to cause an overflow error.  
This doesn’t necessarily happen immediately as you start training the network.

Sometimes your network can be happily training for hours when suddenly the loss function returns NaN and your network has exploded. This can be incredibly annoying, especially when the network has seemingly been training well for a long time.

To prevent this from happening, you need to understand the root cause of the exploding gradient problem.  
One of the reasons for scaling input data into a neural network is to ensure a stable start to training over the first few iterations.

Since the weights of the network are initially randomized, unscaled input could potentially create huge activation values that immediately lead to exploding gradients.

For example, instead of passing pixel values from 0–255 into the input layer, we usually scale these values to between –1 and 1.

Because the input is scaled, it’s natural to expect the activations from all future layers to be relatively well scaled as well. Initially, this may be true, but as the network trains and the weights move further away from their random initial values, this assumption can start to break down. This phenomenon is known as covariate shift

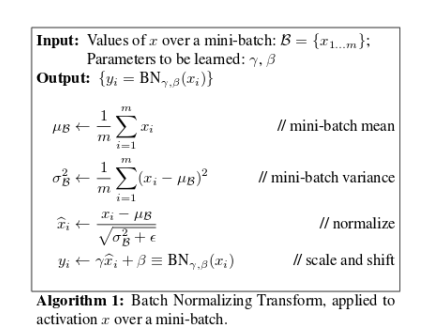
To remain stable, when the network updates the weights, each layer implicitly assumes that the distribution of its input from the layer beneath is approximately consistent across iterations.

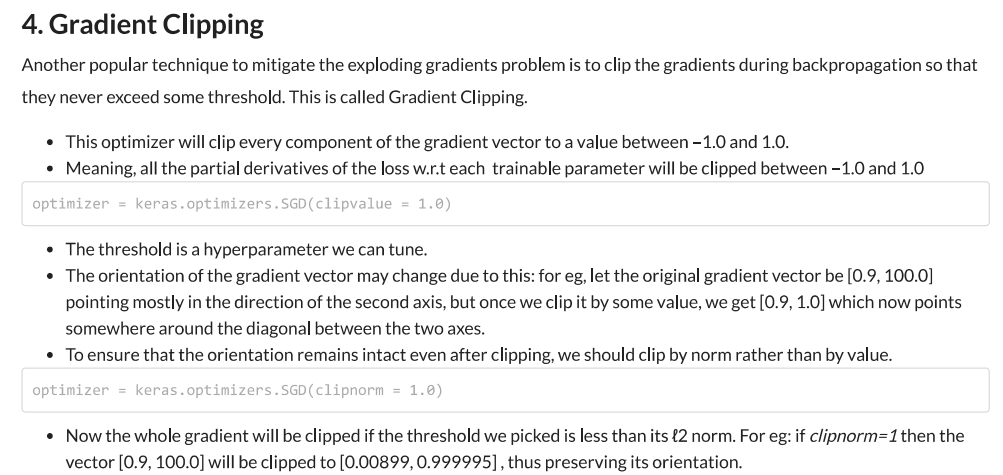
However, since there is nothing to stop any of the activation distributions  
shifting significantly in a certain direction, this can sometimes lead to runaway weight values and an overall collapse of the network.

Batch normalization is a solution that drastically reduces this problem.

The solution is surprisingly simple. A batch normalization layer calculates the mean and standard deviation of each of its input channels across the batch and normalizes by subtracting the mean and dividing by the standard deviation

There are then two learned parameters for each channel, the scale (gamma) and shift (beta). The output is simply the normalized input, scaled by gamma and shifted by beta. Figure 2-15 shows the whole  
process.





**Dropout Layers**  
When studying for an exam, it is common practice for students to use past papers and sample questions to improve their knowledge of the subject material. Some students try to memorize the answers to these questions, but then come unstuck in the exam because they haven’t truly understood the subject matter. The best students use the practice material to further their general understanding, so that they are still able to answer correctly when faced with new questions that they haven’t seen before.

The same principle holds for machine learning.

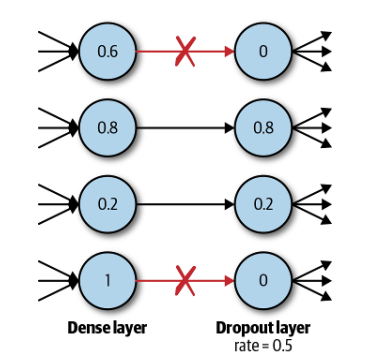
Any successful machine learning algorithm must ensure that it generalizes to unseen data, rather than simply remembering the training dataset. If an algorithm performs well on the training dataset, but not the test dataset, we say that it is suffering from overfitting.

To counteract this problem, we use regularization techniques, which ensure that the model is penalized if it starts to overfit.  
There are many ways to regularize a machine learning algorithm, but for deep learning, one of the most common is by using dropout layers.

This idea was introduced by  
Geoffrey Hinton in 2012 and presented in a 2014 paper by Srivastava et al

Dropout layers are very simple.

During training, each dropout layer chooses a random set of units from the preceding layer and sets their output to zero, as shown in Figure 2-16.





This simple addition drastically reduces overfitting, by ensuring that the  
network doesn’t become overdependent on certain units or groups of units that, in effect, just remember observations from the training set.

If we use dropout layers, the network cannot rely too much on any one unit and therefore knowledge is more evenly spread across the whole network.

This makes the model much better at generalizing to unseen data, because the network has been trained to produce accurate predictions even under unfamiliar conditions, such as those caused by dropping random  
units.

There are no weights to learn within a dropout layer, as the units to drop are decided stochastically.

At test time, the dropout layer doesn’t drop any units, so that the full network is used to make predictions.